HOCHSCHULE RHEIN-WAAL
&
FLUXANA GMBH & CO. KG

BACHELOR'S THESIS

Development of an automated powder dosing system using a 6-DOF collaborative robotic arm (cobot)

by investigating the influence of vibration, angle of dosing, and rotational speed to the mass flow of the powder.

Author:
Abdelrahman MOSTAFA
Matriculation No. 29528

Supervisor:
Prof. Dr. Ronny HARTANTO
Dr. Rainer SCHRAMM

A thesis submitted in fulfillment of the requirements for the Bachelor degree of Science

in the

Mechatronic Systems Engineering
Faculty of Technology & Bionics

October 18, 2023

Declaration of Authorship

I, Abdelrahman MOSTAFA, declare that this thesis titled, "Development of an automated powder dosing system using a 6-DOF collaborative robotic arm (cobot)" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

HOCHSCHULE RHEIN-WAAL

Abstract

Faculty of Technology & Bionics
Research & Development at Fluxana GmbH & Co. KG

Bachelor of Science

Development of an automated powder dosing system using a 6-DOF collaborative robotic arm (cobot)

by Abdelrahman MOSTAFA

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

Declaration of Authorship iii

Abstract vii

Acknowledgements ix

1 Introduction 1
1.1 Motivation . . . 1
1.2 Objectives . . . 1
1.3 Thesis Structure . . . 1

I Basics of Robotics, ROS, and Powder Dosage 3

2 Basics of Robotics 5
2.1 Types of Robots . . . 5
2.2 Robot Components . . . 7
2.2.1 Link . . . 8
2.2.2 Joint . . . 8
2.2.3 Manipulator . . . 8
2.2.4 Wrist . . . 8
2.2.5 End-Effector . . . 8
2.2.6 Actuator . . . 8
2.2.7 Sensors . . . 8
2.2.8 Controller . . . 8
2.3 Robot Kinematics . . . 8
2.4 Robot Control (Software) . . . 8

3 Robot Operating System | ROS 9
3.1 Linux for Robotics . . . 9
3.1.1 What Is Ubuntu? and Why for Robotics? . . . 9
3.1.2 Ubuntu File Structure . . . 10
3.2 Philosophy Behind ROS . . . 11
3.2.1 Peer-to-Peer . . . 11
3.2.2 Tool-Oriented . . . 11
3.2.3 Multilingual . . . 12
3.2.4 Thin . . . 12
3.2.5 Open Source . . . 12
3.3 Preliminaries . . . 12
3.3.1 ROS-Graph [19] . . . 13
3.3.2 Roscore [18] . . . 14
3.3.3 catkin, Workspaces, and ROS Packages . . . 15
3.4 ROS Communication . . . 16

3.4.1 Publishers-Subscribers . . . 16
3.4.2 Services . . . 16
3.4.3 Actions . . . 16
3.5 MoveIt! [24] . . . 16

4 Basics of Powder Dosing **19**
4.1 Affecting Parameters . . . 19

II Experimental Set-up, Methodology, and Results **21**

5 Experimental Set-up **23**
5.1 Frame . . . 23
5.2 Crucibles . . . 23
5.3 Balance . . . 23
5.3.1 Hardware . . . 23
5.3.2 Software . . . 23
5.4 Ned2 Collaborative Robot . . . 23
5.4.1 Hardware Configurations . . . 23
5.4.2 Software Tools . . . 24
5.5 Precision Validation . . . 24
5.6 Vibration Motor . . . 24

6 Methodology **25**
6.1 Sequence Logic . . . 25
6.2 Each State of the State Diagram . . . 25

7 Evaluation & Results **27**
7.1 Evaluation . . . 27

8 Conclusion & Future Work **29**
8.1 Future Work . . . 29

A Frequently Asked Questions **31**
A.1 How do I change the colors of links? . . . 31

B FX_ROS.py Library in Python **33**

Bibliography **41**

List of Figures

2.1 Figure (a) illustrates an instance of a humanoid robot developed by NASA, while Figure (b) exemplifies a cobot. 6

3.1 Ubuntu file system structure 10
3.2 Graphical representation of a ROS system 14
3.3 `roscore` establishes ephemeral connections with the other nodes in the system. 15

List of Tables

2.1 Robots Classification 7

Chapter 1

Introduction

1.1 Motivation

1.2 Objectives

1.3 Thesis Structure

Chapters – this is the folder where you put the thesis chapters. A thesis usually has about six chapters, though there is no hard rule on this. Each chapter should go in its own separate `.tex` file and they can be split as:

- Chapter 1: Introduction to the thesis topic
- Chapter 2: Background information and theory
- Chapter 3: (Laboratory) experimental setup
- Chapter 4: Details of experiment 1
- Chapter 5: Details of experiment 2
- Chapter 6: Discussion of the experimental results
- Chapter 7: Conclusion and future directions

This chapter layout is specialised for the experimental sciences, your discipline may be different.

Guide written by —
Sunil Patel: www.sunilpatel.co.uk
Vel: LaTeXTemplates.com

Part I

Basics of Robotics, ROS, and Powder Dosage

Chapter 2

Basics of Robotics

As an academic field, robotics emerges as a relatively youthful discipline, characterized by profoundly ambitious objectives, the most paramount of which is the creation of machines capable of emulating human behavior and cognitive processes. This quest to engineer intelligent machines inherently compels us to embark on a journey of self-exploration. It prompts us to scrutinize the intricacies of our own design—why our bodies possess the configurations they do, how our limbs synchronize in movement, and the mechanisms behind our acquisition and execution of intricate tasks. The realization that the fundamental inquiries in robotics are intrinsically linked to inquiries about our own existence forms a captivating and immersive aspect of the robotics pursuit. [11]

Robotics is the scientific field dedicated to the study of robots—machines capable of autonomous operation, carrying out various tasks without direct human intervention. While science fiction often envisions robots in humanoid or android forms, real-world robots, especially those designed for industrial applications, typically deviate from human physical resemblance. These robots typically comprise three fundamental components: a mechanical structure, often represented by a robotic arm, enabling physical interaction with the robot's environment or itself; sensors that collect data on various physical attributes such as sound, temperature, motion, and pressure; and a processing system that interprets data from the robot's sensors, providing instructions for task execution.

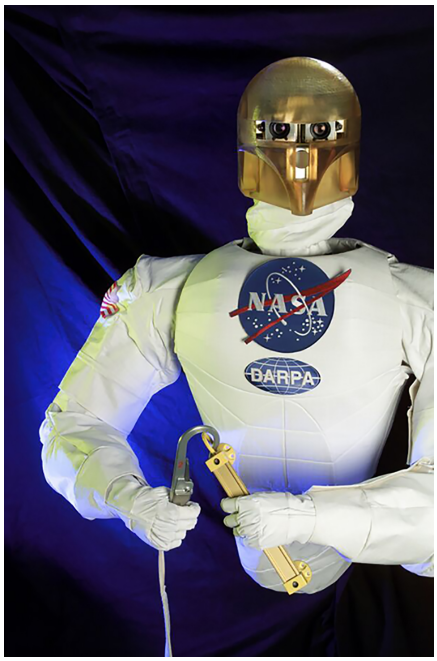
It's worth noting that certain devices, like web-crawling search engine bots that systematically explore the internet to collect information on links and online content, may lack physical mechanical elements. Nonetheless, they are still classified as robots because they exhibit the ability to perform repetitive tasks autonomously.

This chapter delves into an exploration of various robot classifications, delving into the foundational principles of mechanics and kinematics. It also scrutinizes the intricacies of planning and control within the context of collaborative robots (cobots).

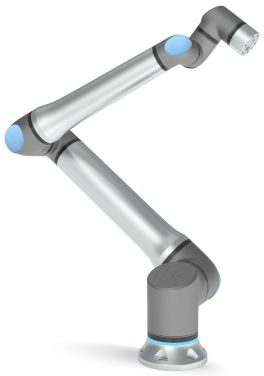
The knowledge presented in this chapter draws significant inspiration from two primary sources, namely '**Modern Robotics**' [11] by Kevin M. Lynch and Frank C. Park, and '**Theory of Applied Robotics**' [9] authored by Professor Reza N. Jazar. For a deeper understanding of these topics, I encourage you to refer to these texts.

2.1 Types of Robots

Across diverse industries, robotics solutions have emerged as catalysts for heightened productivity, elevated safety standards, and increased operational adaptability.



(a) NASA Robonaut [1] [2]



(b) Universal Robot (UR20) [25]

FIGURE 2.1: Figure (a) illustrates an instance of a humanoid robot developed by NASA, while Figure (b) exemplifies a cobot.

Organizations at the vanguard of innovation are discerning forward-looking applications of robotics that yield palpable and quantifiable outcomes. Intel collaborates closely with manufacturers, system integrators, and end-users, actively contributing to the realization of robots that deliver impactful, human-centered results.

According to an article from Intel regarding the classification of robots [8], the current generation of robots has been categorized into six distinct groups.

Autonomous Mobile Robots (AMRs) [5] AMRs navigate their environments and make rapid decisions on the fly. These robots employ advanced technologies like sensors and cameras to gather data from their surroundings. Equipped with onboard processing capabilities, they analyze this data and make well-informed decisions—whether it involves avoiding an approaching human worker, selecting the exact parcel to pick, or determining the suitable surface for disinfection. These robots are self-sufficient mobile solutions that operate with minimal human intervention. [20]

Automated Guided Vehicles (AGVs) [26] While AMRs navigate their surroundings autonomously, AGVs typically operate along fixed tracks or predetermined paths and frequently necessitate human supervision. AGVs find extensive application in scenarios involving the transportation of materials and goods within controlled settings like warehouses and manufacturing facilities.

Humanoids [10] While numerous mobile humanoid robots could, in a technical sense, be classified as Autonomous Mobile Robots (AMRs), this categorization primarily applies to robots fulfilling human-centric roles, frequently adopting human-like appearances. These robots leverage a similar array of technological components as AMRs to perceive, strategize, and execute tasks, encompassing activities such as offering navigational assistance or providing concierge services.

Hybrids [23] Diverse categories of robots are frequently integrated to engineer hybrid solutions that possess the capacity to execute intricate operations. For instance, the fusion of an AMR with a robotic arm can yield a versatile system tailored for the handling of packages within a warehouse environment. As functionalities are amalgamated within single solutions, there is a concurrent consolidation of computational capabilities.

Articulated Robots [22] Commonly referred to as robotic arms, are designed to replicate the versatile functions of the human arm. These systems typically incorporate a range of rotary joints, varying from two to as many as ten. The inclusion of additional joints or axes equips these robotic arms with a wider range of motion capabilities, rendering them particularly well-suited for tasks such as arc welding, material manipulation, machine operation, and packaging.

Cobots [14] Collaborative Robots, commonly referred to as cobots, are engineered with the specific purpose of working in tandem with, or directly alongside, human operators. Unlike many other categories of robots that function autonomously or within strictly segregated workspaces, cobots share work environments with human personnel to enhance their collective productivity. Their primary role often involves the removal of manual, hazardous, or physically demanding tasks from daily operations. In certain scenarios, cobots are capable of responding to and learning from human movements, further enhancing their adaptability.

The initial four robots fall under the category of mobile robots, possessing the capability to navigate within their surroundings, while the latter two are categorized as stationary robots, as detailed in table 2.1 below.

TABLE 2.1: Robots Classification.

Mobile	**Stationary**
AMRs	
AGVs	Articulated robots
Humanoids	**Cobots**
Hybrids	

Within the scope of this paper, our exclusive focus will be on **cobots** [14]. Across all the experiments conducted in this study, a cobot (Ned2, detailed and described in chapter 5) has been consistently utilized.

2.2 Robot Components

In our study, we establish a kinematic model for a robotic manipulator, which is essentially a multi-body system comprising interconnected rigid bodies. These bodies are connected through revolute or prismatic joints, enabling relative movement. We employ principles of rigid body kinematics to elucidate the relative motions between these interconnected bodies.

It's imperative to note that a comprehensive robotic system encompasses not only

the manipulator or rover but also components such as the wrist, end-effector, actuators, sensors, controllers, processors, and software. [9]

2.2.1 Link

In the realm of robotics, each individual rigid component within a robot that possesses the capacity to move concerning all other components is formally known as a 'link.' This terminology accommodates various descriptions, including 'bar,' 'arm,' or any object deemed equivalent to a link in the context of robot mechanics. A robot arm or link, in essence, represents a solid, rigid element capable of relative motion when compared to the other links within the robotic structure.

Moreover, when we encounter two or more linked components that are entirely constrained in terms of relative movement, they are collectively regarded as a 'compound link,' forming a unified and motionally inseparable entity within the robot's framework.

2.2.2 Joint

2.2.3 Manipulator

2.2.4 Wrist

2.2.5 End-Effector

2.2.6 Actuator

2.2.7 Sensors

2.2.8 Controller

2.3 Robot Kinematics

2.4 Robot Control (Software)

Chapter 3

Robot Operating System | ROS

3.1 Linux for Robotics

Linux is a free, open-source operating system that includes several utilities that will significantly simplify your life as a robot programmer. As will be shown in the upcoming sections, ROS (Robot Operating System) is based on a Linux system. All commands and concepts explained here are taken from the Linux tutorial made by the University of Surrey. [**linuxforrobotics**]

3.1.1 What Is Ubuntu? and Why for Robotics?

Ubuntu, accessible at www.ubuntu.com, stands as a widely acclaimed Linux distribution rooted in the Debian architecture (source: https://en.wikipedia.org/wiki/Debian). Notably, it's freely available and open source, permitting extensive customization for specific applications. Ubuntu boasts an extensive software repository, comprising over 1,000 software components, encompassing essentials such as the Linux kernel, GNOME/KDE desktop environments, and a suite of standard desktop applications, including word processing tools, web browsers, spreadsheets, web servers, programming languages, integrated development environments (IDEs), and even PC games. Versatile in its deployment, Ubuntu can operate on both desktop and server platforms, accommodating architectures like Intel x86, AMD-64, ARMv7, and ARMv8 (ARM64). Canonical Ltd., headquartered in the UK (www.canonical.com), provides substantial backing to Ubuntu.

In the realm of robotics, software stands as the nucleus of any robotic system. An operating system serves as the foundation, facilitating seamless interaction with robot actuators and sensors. A Linux-based operating system, such as Ubuntu, offers unparalleled flexibility in interfacing with low-level hardware while affording provisions for tailored OS configurations tailored to specific robot applications. Ubuntu's merits in this context are manifold: it exhibits responsiveness, maintains a lightweight profile, and upholds stringent security measures. Additionally, Ubuntu boasts a robust community support ecosystem and a cadence of frequent releases, ensuring its perpetual relevance. It also offers long-term support (LTS) releases, guaranteeing user assistance for up to five years. These compelling attributes have cemented Ubuntu as the preferred choice among developers in the Robot Operating System (ROS) community. Indeed, Ubuntu stands as the sole operating system that enjoys comprehensive support from ROS developers. The Ubuntu-ROS synergy emerges as the quintessential choice for programming robots. [**linuxforrobotics**]

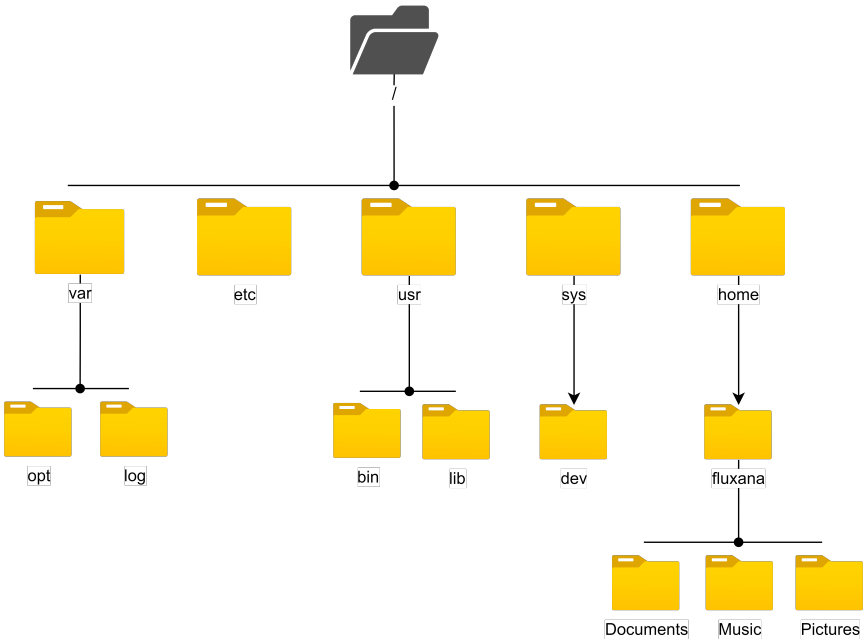


FIGURE 3.1: Ubuntu file system structure

3.1.2 Ubuntu File Structure

Similar to the 'C drive' in a Windows operating system, Linux incorporates a dedicated storage area for its system files, known as the root file system. This root file system is established during the Ubuntu installation process, with the assignment of '/' as its designated mount point. For a visual representation of the Ubuntu file system architecture, refer to Figure 3.1.

The following describes the uses of each folder in the file system:

- */bin and /sbin*: These directories house essential system applications, akin to the 'C:/Windows' folder in Windows.
- */etc*: Within this directory, system configuration files are stored.
- */home/yourusername*: Equivalent to the 'C:/Users' directory in Windows, this directory serves as the user's home.
- */lib*: Similar to '.dll' files in Windows, the '/lib' directory contains library files.
- */media*: This directory serves as the mount point for removable media.
- */root*: The '/root' directory contains files associated with the root user, who holds administrative privileges in the Linux system.
- */usr*: Pronounced 'user,' the '/usr' directory hosts a majority of program files, akin to 'C:/Program Files' in Microsoft Windows.
- */var/log*: Within this directory, you'll find log files generated by various applications.
- */home/yourusername/Desktop*: The location for Ubuntu desktop files.
- */mnt*: Mounted partitions are accessible in this directory.

- `/boot`: This directory stores essential files required for the boot process.
- `/dev`: Linux device files are located here.
- `/opt`: The '/opt' directory serves as the designated location for optionally installed programs. (For instance, ROS is installed in '/opt').
- `/sys`: This directory houses files containing critical information about the system.

3.2 Philosophy Behind ROS

The philosophical objectives of ROS can be succinctly described as follows [16]:

- Decentralized collaboration: Emphasizing **peer-to-peer** interactions.
- **Tool-oriented** approach: Focusing on the development of a robust set of tools.
- **Multilingual support**: Enabling compatibility with multiple programming languages.
- **Thin** design: Prioritizing a streamlined framework.
- Openness and freedom: Being freely available and based on **open-source** principles.

To the best of our knowledge, no existing framework encompasses this specific set of design principles. This section aims to delve into these philosophies, elucidating how they have profoundly influenced the design and implementation of ROS [16].

3.2.1 Peer-to-Peer

ROS comprises a multitude of compact software components that establish connections among themselves, facilitating a perpetual exchange of messages. These messages are transmitted directly from one software component to another without the need for a centralized routing service. While this architecture may introduce added complexity to the underlying system infrastructure, it yields a critical advantage: scalability. As data volume increases, the ROS system can efficiently accommodate the rising demands without compromising its performance.

3.2.2 Tool-Oriented

Drawing inspiration from the enduring architectural principles of Unix, ROS exemplifies how intricate software systems can be constructed from an assembly of numerous small, versatile programs. Distinguishing itself from many other robotics software frameworks, ROS does not adopt a singular, integrated development and run-time environment. Instead, it delegates various tasks—such as source code tree navigation, system visualization (refer to section 3.3.1), graphical data plotting, documentation generation, data logging, and more—to discrete software programs. This decentralized approach encourages the continuous refinement and evolution of these tools. Ideally, users have the flexibility to substitute existing tools with improved implementations tailored to specific task domains.

In recent ROS iterations, there has been an advancement where multiple tools can be seamlessly integrated into single processes to enhance operational efficiency

or facilitate user-friendly interfaces for operators and debugging. Nevertheless, the foundational principle persists: individual tools remain compact and versatile in nature.

3.2.3 Multilingual

Software tasks vary in their complexity and requirements, sometimes favoring 'high-productivity' scripting languages like Python or Ruby, while other scenarios demand the efficiency of faster languages such as C++. Preferences for languages like Lisp or MATLAB [4] also come into play, sparking debates on the best-suited language for specific tasks. Recognizing the value of these diverse perspectives and the contextual utility of languages, ROS adopts a multilingual approach. ROS permits software modules to be written in a wide array of languages, provided that a compatible client library exists. As of the current writing, ROS supports client libraries for C++, Python, LISP, Java, JavaScript, MATLAB, Ruby, Haskell, R, Julia, and others. This book predominantly utilizes the Python client library for code examples, balancing space considerations and user-friendliness. However, it's important to note that the tasks discussed here can be achieved using any of the available client libraries.

3.2.4 Thin

The conventions within ROS promote a development approach where contributors create independent libraries and subsequently integrate them with ROS modules to facilitate message exchange. This intermediary layer serves a dual purpose: it allows for the versatile reuse of software in contexts beyond ROS, and it streamlines the process of conducting automated tests through established continuous integration tools.

3.2.5 Open Source

The core of ROS operates under the permissive BSD license [17], allowing both commercial and noncommercial utilization. ROS employs interprocess communication (IPC) to facilitate data exchange between modules. This approach enables systems built with ROS to have flexible licensing arrangements for their components. Commercial systems, for instance, may incorporate a combination of closed-source and open-source modules, while academic and hobby projects often adhere to open-source principles. Additionally, commercial product development frequently takes place within closed environments. The ROS license accommodates these diverse use cases and remains fully compliant with each scenario. [16]

3.3 Preliminaries

Before delving into ROS, it's essential to introduce the fundamental concepts that underpin this framework. ROS systems are constituted by a multitude of autonomous programs that maintain continuous communication with one another. This section provides an in-depth exploration of this architectural setup and the associated command-line tools. It further delves into the intricate aspects of ROS naming conventions and namespaces, demonstrating their role in facilitating code reusability. [15]

3.3.1 ROS-Graph [19]

One of the original challenges inspiring the creation of ROS was commonly known as the 'fetch an item' problem. This scenario involved a relatively large and complex robot equipped with various sensors, a manipulator arm, and a mobile base. In the 'fetch an item' problem, the robot's objective is to navigate a typical home or office environment, locate a specified item, and transport it to the designated location. This task led to several key observations, which subsequently became foundational design goals for ROS:

- The application task can be broken down into numerous autonomous subsystems, encompassing areas like navigation, computer vision, and grasping.
- These subsystems are adaptable for various tasks, such as security patrols, cleaning, and mail delivery, among others.
- By implementing appropriate hardware and geometry abstraction layers, the majority of application software can be made compatible with different robotic platforms.

These principles are exemplified through the core structure of a ROS system: its graphical representation. In ROS, multiple programs operate concurrently and communicate by exchanging messages. This system structure is conveniently portrayed as a mathematical graph, with nodes representing individual programs and edges indicating their communication. While Figure 3.2 illustrates a sample ROS graph from one of the early 'fetch an item' implementations, the specific details are less significant compared to the overarching concept of a ROS system as an assembly of nodes engaged in message-based communication. This representation serves as a practical framework for software development, emphasizing the modular nature of ROS programs, or 'nodes,' as integral components within a larger system.

In summary, within a ROS graph, a node signifies a software module engaged in message transmission, and an edge denotes the flow of messages between two nodes. While complexity can increase, nodes are typically POSIX processes, and edges are akin to TCP connections, enhancing fault tolerance as a software crash typically affects only the crashing process, leaving the rest of the graph operational. The events leading to the crash can often be reconstructed by logging messages entering a node and replaying them within a debugger at a later time.

One of the most significant advantages of a loosely coupled, graph-based architecture is the capacity to rapidly prototype complex systems with minimal or no need for additional 'glue' software during experimentation. Individual nodes, such as the object recognition node in a 'fetch an item' system, can be effortlessly replaced by launching an entirely different process that handles images and generates labeled objects. Beyond node replacement, entire segments of the graph (subgraphs) can be dynamically dismantled and substituted with other subgraphs in real time. This flexibility extends to replacing real-robot hardware drivers with simulators, swapping navigation subsystems, fine-tuning algorithms, and more. Since ROS dynamically generates the necessary network backends, the entire system fosters an interactive environment that encourages experimentation.

To this point, we have assumed that nodes discover each other, but we have not elaborated on the process. Amidst the extensive network traffic, how do nodes locate and initiate message exchange? The solution lies in a program known as '`roscore`'.

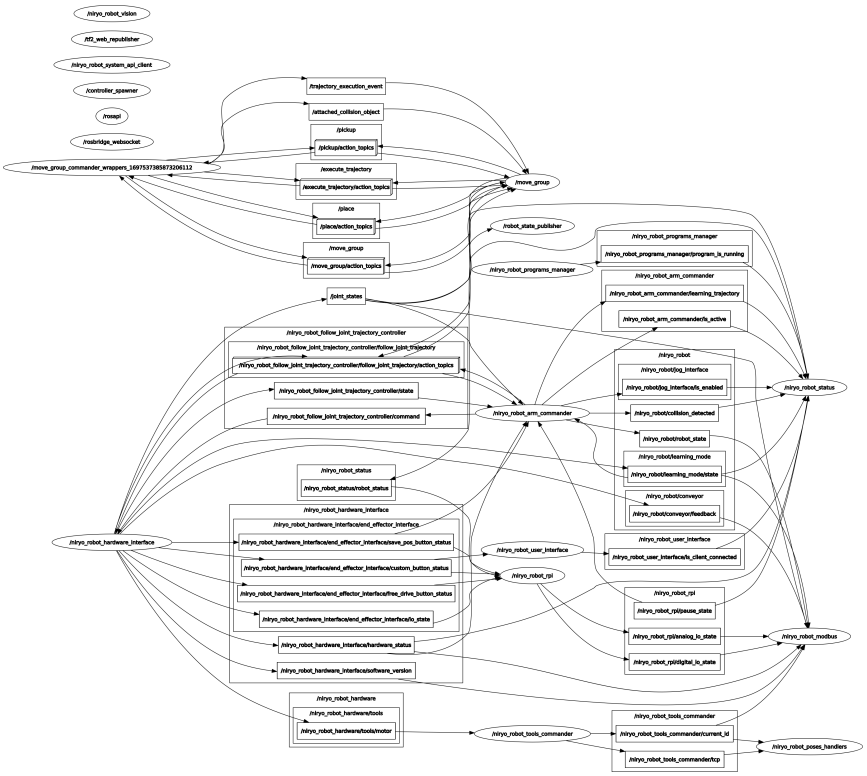


FIGURE 3.2: Graphical representation of a ROS system for 'Niryo-Ned2 [7]' robot—nodes/topics within the graph symbolize individual software modules, while edges denote message streams facilitating the exchange of sensor data, actuator commands, planner states, intermediate representations, and other relevant information.

3.3.2 Roscore [18]

`roscore` serves as a vital component within the ROS ecosystem by facilitating connections between nodes to enable message transmission. During initialization, each node registers its published message streams and desired subscriptions with `roscore`, allowing it to establish direct peer-to-peer connections with other nodes participating in the same message topics. A functioning `roscore` is imperative for any ROS system since it serves as a vital reference point for nodes to discover one another.

It's important to note that while `roscore` plays a crucial role in aiding nodes in locating their peers, the actual message transmission between nodes occurs in a peer-to-peer manner. This setup can sometimes be misconstrued, especially for individuals accustomed to client/server systems from web-based backgrounds, wherein the roles of clients and servers are more distinct. The ROS architecture, however, functions as a hybrid system, integrating aspects of both client/server and fully distributed models, thanks to the central role of `roscore`, which acts as a naming service for peer-to-peer message streams.

When a ROS node initiates, it relies on the presence of an environment variable, ROS_MASTER_URI, which should contain a URL of the form `http://hostname:11311/`. This URL signifies the existence of a functioning `roscore` accessible on port 11311, hosted on a machine named hostname, which can be reached over the network.

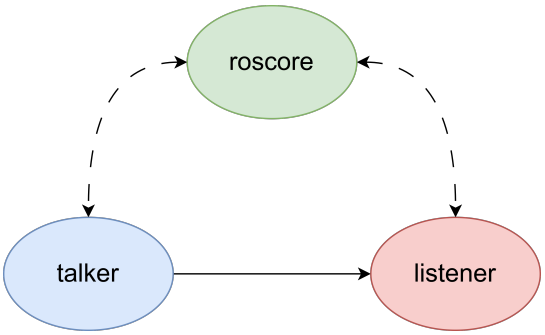


FIGURE 3.3: `roscore` establishes ephemeral connections with the other nodes in the system.

With this information, nodes communicate with `roscore` at startup to register themselves and query for other nodes and message streams by name. Each node informs `roscore` about the messages it can provide and those it wishes to subscribe to. `roscore`, in return, supplies the necessary details about the message producers and consumers. In graphical terms, every node within the system can periodically utilize `roscore` services to identify and connect with its peers. This is illustrated by the dashed lines in Figure 3.3, signifying that in a basic two-node setup, the talker and listener nodes intermittently make service calls to `roscore` while directly engaging in peer-to-peer message exchange.

`Roscore` also serves as a parameter server, extensively utilized by ROS nodes for configuration purposes. It enables nodes to store and retrieve various data structures, including robot descriptions and algorithm parameters. To interact with the parameter server, ROS provides a command-line tool called '`rosparam`,' which we will use throughout this book.

We will delve into examples of using roscore shortly. For now, it's essential to remember that roscore facilitates nodes in discovering other nodes. Before we proceed to run some nodes, it's worth understanding how ROS organizes packages and gaining some insight into the ROS build system, known as 'catkin. [15]

3.3.3 catkin, Workspaces, and ROS Packages

`Catkin` serves as the ROS build system, comprising a set of tools utilized by ROS for generating executable programs, libraries, scripts, and interfaces that can be employed by other code. If you are developing your ROS code in C++, a good understanding of catkin is essential. However, since this book employs Python for its examples, we won't delve deeply into its intricacies. Nevertheless, we will explore its basic functionalities to some extent. For those interested in a more comprehensive understanding, the catkin wiki page is an excellent resource. If you are curious about why ROS has its dedicated build system, you can refer to the catkin conceptual overview wiki page. To install ros-melodic catkin workspace, you can use the following command:

Command

```
sudo apt-get install ros-melodic-catkin
```

ROS Catkin Workspace

The ros workspace has several folders. Following, we will be looking at the function of each folder.

src Forlder The 'src' directory within the catkin workspace serves as the designated location for creating or importing new packages from repositories. It's important to note that ROS packages are only built and turned into executables when they reside in the 'src' directory. When the 'catkin_make' command is executed from the workspace directory, it scans the 'src' folder, building each package found there.

build Folder When the 'catkin_make' command is executed within the ROS workspace, the catkin tool generates certain build files and intermediate CMake cache files within the 'build' directory. These cache files play a crucial role in preventing the need to rebuild all packages each time you run 'catkin_make.' For example, if you initially build five packages and subsequently introduce a new package to the 'src' folder, only the new package will be built during the next 'catkin_make' command. This efficiency is achieved through the utilization of cache files within the 'build' directory. It's important to note that deleting the 'build' folder will trigger a complete rebuild of all packages.

devel Folder When 'catkin_make' is executed, it triggers the build process for each package, resulting in the creation of target executables if the build is successful. These executables are saved within the 'devel' folder, which contains shell script files designed to incorporate the current workspace into the ROS workspace path. Access to the packages within the current workspace is only enabled when this script is executed. Typically, the following command is employed for this purpose.

Command

```
source ~/<workspace_name>/devel/setup.bash
```

3.4 ROS Communication

3.4.1 Publishers-Subscribers

3.4.2 Services

3.4.3 Actions

3.5 MoveIt! [24]

MoveIt![3] serves as the primary software framework within the Robot Operating System (ROS) for motion planning and mobile manipulation. It has garnered acclaim for its seamless integration with various robotic platforms, including the PR2 [27], Robonaut [1], and DARPA's Atlas robot. MoveIt! is

primarily coded in C++, augmented by Python bindings to facilitate higher-level scripting. Embracing the fundamental principle of software reuse, advocated for in the realm of robotics [12], MoveIt! adopts an agnostic approach towards robotic frameworks, such as ROS. This approach entails a formal separation between its core functionality and framework-specific elements, ensuring flexibility and adaptability, especially in inter-component communication.

By default, MoveIt! leverages the core ROS build and messaging systems. To facilitate effortless component swapping, MoveIt! extensively employs plugins across its functionality spectrum. This includes motion planning plugins (currently utilizing OMPL), collision detection (presently incorporating the Fast Collision Library (FCL) [13]), and kinematics plugins (employing the OROCOS Kinematics and Dynamics Library (KDL) [21] for both forward and inverse kinematics, accommodating generic arms alongside custom plugins).

MoveIt!'s principal application domain lies in manipulation, encompassing both stationary and mobile scenarios, across industrial, commercial, and research settings. For a more comprehensive exploration of MoveIt!, interested readers are encouraged to refer to **Cite here**.

Chapter 4

Basics of Powder Dosing

4.1 Affecting Parameters

Part II

Experimental Set-up, Methodology, and Results

Chapter 5

Experimental Set-up

This chapter initiates a comprehensive exploration of the experimental setup. It commences with an overview of the workspace frame, encompassing the array of interconnected devices. These include various crucibles, the balance device, both its hardware and software components, the Ned2-cobot with its hardware configurations and software tools, precision validation procedures and concludes with an examination of the vibration motor, which is an auxiliary component integrated with the cobot.

5.1 Frame

5.2 Crucibles

5.3 Balance

5.3.1 Hardware

5.3.2 Software

5.4 Ned2 Collaborative Robot

Ned2 is a collaborative robot, often referred to as a cobot, developed by the French company Niryo [6]. This particular cobot has been purpose-built for educational and research applications, serving as a valuable tool for the development of proof of concepts and experimental work. In the context of this research, the Ned2 cobot played a pivotal role in conducting experiments.

The forthcoming sections delve into an in-depth examination of the hardware specifications and software options offered by the Ned2 cobot.

5.4.1 Hardware Configurations

Ned2 is a six-axis collaborative robot, based on open-source technologies. It is intended for education, research and Industry 4.0." [7]

Incorporating the same aluminum framework as its predecessor, Ned2 maintains its commitment to meeting your exacting standards in terms of durability, precision, and repeatability (with an accuracy, and a repeatability of 0.5 mm).

Ned2 operates on the Ubuntu 18.04 platform and utilizes the ROS Melodic

framework, capitalizing on the capabilities of the **Raspberry Pi 4**. This high-performance **64-bit ARM V8 processor**, coupled with **4GB of RAM**, empowers Ned2 to deliver enhanced performance.

This iteration of Ned2 introduces advanced servo motors equipped with Silent Stepper Technology, significantly reducing the operational noise of the robot.

The technical specifications of Ned2 are described as shown in ..... table below.

5.4.2 Software Tools

Ned2 represents a collaborative robot, hinging on the Ubuntu 18.04 platform and ROS (Robot Operating System) Melodic—a widely adopted open-source solution in the field of robotics. Leveraging ROS, Ned2 offers an extensive array of libraries that empower users to create a wide spectrum of programs, from the simplest to the most intricate, thus ensuring adaptability to diverse operational requirements. [7]

5.5 Precision Validation

5.6 Vibration Motor

Chapter 6

Methodology

6.1 Sequence Logic

6.2 Each State of the State Diagram

Chapter 7

Evaluation & Results

7.1 Evaluation

Chapter 8

Conclusion & Future Work

small description of what I have done.. What are my final findings and thoughts...

The future work, what to come.

8.1 Future Work

Appendix A

Frequently Asked Questions

A.1 How do I change the colors of links?

The color of links can be changed to your liking using:

`\hypersetup{urlcolor=red}`, or
`\hypersetup{citecolor=green}`, or
`\hypersetup{allcolor=blue}`.

If you want to completely hide the links, you can use:

`\hypersetup{allcolors=.}`, or even better:
`\hypersetup{hidelinks}`.

If you want to have obvious links in the PDF but not the printed text, use:

`\hypersetup{colorlinks=false}`.

Appendix B

FX_ROS.py Library in Python

```
#!/usr/bin/env python
import tf
import time
import sys

import numpy as np
import matplotlib.pyplot as plt

import rospy
#from ActionClient import ActionClient
from sensor_msgs.msg import JointState
from niryo_robot_arm_commander.srv import GetFK, GetFKRequest,
    GetJointLimits, JogShift, JogShiftRequest, JogShiftResponse
#from niryo_robot_msgs.srv import SetBool, SetBoolRequest, SetInt,
    SetIntRequest, Trigger

#from niryo_robot_arm_commander.msg import ArmMoveCommand,
    RobotMoveGoal, RobotMoveAction

from moveit_msgs.srv import GetPositionFK, GetPositionIK

from std_msgs.msg import Header
from moveit_msgs.msg import RobotState as RobotStateMoveIt

from geometry_msgs.msg import Pose
import geometry_msgs
from niryo_robot_msgs.msg import RobotState
import moveit_commander
import moveit_msgs.msg
import actionlib

#robot = moveit_commander.RobotCommander()
#scene = moveit_commander.PlanningSceneInterface()
#global arm
def Connect_to_arm():
    global arm
    try:
        arm = moveit_commander.move_group.MoveGroupCommander("arm")
    except:
        raise RuntimeError

def Call_Aservice(service_name, type, request_name=None, req_args=None)
    :
    """Call a ROS service.

    Parameters:
    ......................
    service_name: str
    type: srv
```

```
    request_name: None (srv)
    req_args: None (dictionary) ex. {'positon': 210, 'id': 11, 'value': False}
    should_return ?: None (int) >> is set to 1, if you want to return the response of the service.

    Returns:
    ......................
    If should_return is set to 1, the function is going to return the response of the service.
    Otherwise, the function should only call the service to do a certain action with no return.
    """
    try:
        rospy.wait_for_service(service_name, 2)
    except (rospy.ServiceException, rospy.ROSException) as e:
        rospy.logerr("Timeout and the Service was not available : " + str(e))
        return RobotState()

    try:
        service_call = rospy.ServiceProxy(service_name, type)

        if request_name == None:
            response = service_call()
        else:
            request = request_name()
            for key, value in req_args.items():
                #print("f{key} = {value}")
                method = setattr(request, key, value)
            response = service_call(request)

    except rospy.ServiceException as e:
        rospy.logerr("Falied to call the Service: " + str(e))
        return 0

    return response

def Subscribe(topic_name, type, msg_args):
    """Subscribe to a certain topic.

    Parameters:
    ......................
    topic_name: str
    type: srv
    msg_args: list >> list of strings, which contains the arguments that we need to read from the topic.

    Returns:
    ......................
    Return a list of the read values from each argument.
    If we have only one argument, it returns the value of this argument only, not a list.
    """

    #rospy.init_node('FX_ROS_Subscriber')

    try:
        msg = rospy.wait_for_message(topic_name, type, 2)
    except:
        rospy.logerr("Timeout and the Topic Did not recieve any messages")
        return 0
```

```
101
102
103     value = []
104
105     if len(msg_args) == 1:
106         value = getattr(msg, msg_args[0])
107     else:
108         for i in msg_args:
109             value.append(getattr(msg, i))
110
111     return value
112
113 def Get_joints():
114     """return a tuple of 6 values for each joint from 1 till 6"""
115
116     joints_values = Subscribe('/joint_states', JointState, ["position"
        ])
117
118     return joints_values
119
120 def get_pose():
121     """Gets the pose values from the robot_state topic.
122     Return:
123     ..................................
124     a list of two dictionaries, the first is positions (x,y,z),
125     whereas the second is the rpy (roll, pitch, yaw)
126     """
127
128     return Subscribe('/niryo_robot/robot_state', RobotState, ['position
        ', 'rpy'])
129
130 def get_pose_list():
131     """Use get_pose() function to get the pose, and turn it into a list
        .
132     Return:
133     ............................
134     A list of floats >>> [x, y, z, roll, pitch, yaw]
135     """
136
137     pose = get_pose()
138     position = pose[0]
139     rpy = pose[1]
140
141     return [position.x, position.y, position.z, rpy.roll, rpy.pitch,
        rpy.yaw]
142
143 def Get_FK_Niryo(joints):
144     """Give the the joints' values to the forward kinematics service
145     provided by Niryo, and get the pose coordinations.
146     """
147     fk_service = '/niryo_robot/kinematics/forward'
148     return Call_Aservice(fk_service, GetFK, GetFKRequest, {'joints':
        joints}, should_return=1).pose
149
150 def FK_Moveit(joints):
151     """Get Forward Kinematics from the MoveIt service directly after
        giving joints
152     :param  joints
153     :type   joints: list of joints values
154     :return: A Pose state object
155     @example of a return
156
157 position:
```

```
  x: 0.278076372862
  y: 0.101870353599
  z: 0.425462888681
orientation:
  x: 0.0257527874589
  y: 0.0122083384395
  z: 0.175399274203
  w: 0.984084775322

    """
    rospy.wait_for_service('compute_fk', 2)
    moveit_fk = rospy.ServiceProxy('compute_fk', GetPositionFK)

    fk_link = ['base_link', 'tool_link']
    header = Header(0, rospy.Time.now(), "world")
    rs = RobotStateMoveIt()

    rs.joint_state.name = ['joint_1', 'joint_2', 'joint_3', 'joint_4',
    'joint_5', 'joint_6']
    rs.joint_state.position = joints

    reponse = moveit_fk(header, fk_link, rs)

    return reponse.pose_stamped[1].pose


def Jog_shift(joints_or_pose, axis, value):
    """Use the service jog_shift_commander to shift one axis.
    Paramters:
    ........................
    joints_or_pose: int >>> 1 for joints_shift, and 2 for pose_shift
    axis: int >>> (1,2,3,4,5,6) = (x,y,z,roll,pitch,yaw)
    value: float >> the value for which you want to shift the Jog axis.

    Returns: None
    ........................
    """

    axis -= 1
    name = "/niryo_robot/jog_interface/jog_shift_commander"
    shift_values = [0, 0, 0, 0, 0, 0]
    shift_values[axis] = value

    req_arg = {'cmd': joints_or_pose, 'shift_values': shift_values}

    Call_Aservice(name, JogShift, JogShiftRequest, req_arg)

def Move_pose_axis(axis, new=None, add=None, arm_speed=None):
    """You should either put a value to add or new, not both.

    Parameters:
    ........................
    * axis: str -> (x, y, z, roll, pitch, or yaw)
    * new: float -> The new coordination you want to give to a certain
    axis.
           "new" will always overwrite the value of the axis.
    * add: float -> the value in meters or radians you want to add to a
     certain axis.
    * arm_speed: float (optional) -> between 0 and 1. (0,1]
    Returns: None
    ........................
    """
    FK = get_pose()
```

```
218     axises = ['x','y','z']
219 
220     pose = Pose()
221     p_goal = pose.position
222     orn_goal = pose.orientation
223 
224     p_current = FK[0]
225 
226     rpy_current = FK[1]
227 
228     if add:
229         if axis.lower() in axises:
230             current_value = getattr(p_current, axis)
231             setattr(p_current, axis, current_value+add)
232         else:
233             current_value = getattr(rpy_current, axis)
234             setattr(rpy_current, axis, current_value+add)
235     if new:
236         if axis.lower() in axises:
237             setattr(p_current, axis, new)
238         else:
239             setattr(rpy_current, axis, new)
240 
241 
242     p_goal.x = p_current.x
243     p_goal.y = p_current.y
244     p_goal.z = p_current.z
245 
246     orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations
    .quaternion_from_euler(rpy_current.roll,rpy_current.pitch,
    rpy_current.yaw)
247 
248     arm.set_pose_target(pose)
249 
250     if arm_speed:
251         set_speed(arm_speed)
252     arm.go(wait=True)
253 
254     arm.stop()
255     arm.clear_pose_targets()
256 
257 def Move_to_pose(pose_values, arm_speed=None):
258     """Move to a given pose values.
259     Parameters:
260     ......................
261 
262     pose_values: list or tuble -> [x, y, z, roll, pitch, yaw]
263     arm_speed: float (optional) -> between 0 and 1. (0,1]
264     """
265 
266     pose = Pose()
267     p_goal = pose.position
268     orn_goal = pose.orientation
269 
270     p_goal.x = pose_values[0]
271     p_goal.y = pose_values[1]
272     p_goal.z = pose_values[2]
273 
274     roll = pose_values[3]
275     pitch = pose_values[4]
276     yaw  = pose_values[5]
277 
```

```
278     orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations
        .quaternion_from_euler(roll,pitch,yaw)
279 
280     #arm.set_goal_tolerance(0.001)
281     if arm_speed:
282         set_speed(arm_speed)
283     arm.set_pose_target(pose)
284     arm.go(wait=True)
285 
286     arm.stop()
287     arm.clear_pose_targets()
288 
289 def move_to_joints(joints, arm_speed=None):
290     """Move to a given joint values.
291     Parameters:
292     ......................
293 
294     joints: list or tuble -> [joint1, joint2, joint3, joint4, joint5,
        joint6]
295     arm_speed: float (optional) -> between 0 and 1. (0,1]
296     """
297     joints_limits = Get_Joints_limits()
298 
299     for i in range(6):
300         if joints_limits.joint_limits[i].max < joints[i] or joints[i] <
         joints_limits.joint_limits[i].min:
301             print("Joint{} = {}, which is out of limit!".format(i+1,
        joints[i]))
302             print("Joint{} can not be more than {} neither less than {}
        ".format(i+1, joints_limits.joint_limits[i].max, joints_limits.
        joint_limits[i].min))
303             return
304         else:
305             pass
306 
307     #arm.set_joint_value_target(joints)
308     if arm_speed:
309         set_speed(arm_speed)
310     arm.go(joints, wait=True)
311 
312     arm.stop()
313 
314 def Move_joint_axis(axis, new=None, add=None, arm_speed=None):
315     """You should either put a value to add or new, not both.
316 
317     Parameters:
318     ......................
319     * axis: int -> the number of the joint that you want to move
320 
321     * new: float -> The new coordination you want to give to a joint (
        axis).
322             "new" will always overright the value of the axis.
323     * add: float -> the value in meters change in a certain joint (axis
        ).
324     * arm_speed: float (optional) -> between 0 and 1. (0,1]
325 
326     Returns: None
327     ......................
328     """
329     moving_joints = list(Get_joints())
330 
331     if new:
332         moving_joints[axis-1] = new
```

```
333     elif add:
334         moving_joints[axis-1] += add
335 
336     joints_limits = Get_Joints_limits()
337 
338     if joints_limits.joint_limits[axis-1].max < moving_joints[axis-1]
    or moving_joints[axis-1] < joints_limits.joint_limits[axis-1].min:
339         print("The joint{} can not be more than {} neither less than {}
    ".format(axis, joints_limits.joint_limits[axis-1].max,
    joints_limits.joint_limits[axis-1].min))
340         return 0
341     else:
342         pass
343 
344     arm.set_joint_value_target(moving_joints)
345     if arm_speed:
346         set_speed(arm_speed)
347     arm.go(moving_joints, wait=True)
348 
349     arm.stop()
350 
351 def Get_Joints_limits():
352     """Getting the limits for each joint.
353 
354     You can get any joint limits as following:
355 
356     Get_Joints_limits().joint_limits[0 - 5].max (float)
357     Get_Joints_limits().joint_limits[0 - 5].min (float)
358     Get_Joints_limits().joint_limits[0 - 5].name (str)
359 
360     Where 0 for (joint 1), and 5 for (joint 6)
361     max, min, or name would give the maximum, minimum, or name of the
    indicated joint.
362     """
363 
364     joints_limits = Call_Aservice('/niryo_robot_arm_commander/
    get_joints_limit', GetJointLimits)
365     return joints_limits
366 
367 def set_speed(speed):
368     """Set a scaling factor for optionally reducing the maximum joint
    velocity. Allowed values are in (0,1]."""
369     arm.set_max_velocity_scaling_factor(speed)
370 
371 def wait(duration):
372     """wait for a certain time.
373 
374     :param duration: duration in seconds
375     :type duration: float
376     :rtype: None
377     """
378     time.sleep(duration)
379 
380 def move_with_action(pose):
381     """Still under development"""
382 
383     moveit_commander.roscpp_initialize(sys.argv)
384     rospy.init_node('simple_action', anonymous=True)
385 
386     robot_arm = moveit_commander.move_group.MoveGroupCommander("arm")
387 
388     robot_client = actionlib.SimpleActionClient('execute_trajectory',
    moveit_msgs.msg.ExecuteTrajectoryAction)
```

```
389     robot_client.wait_for_server()
390     #rospy.loginfo('Execute Trajectory server is available for robot')
391 
392     robot_arm.set_pose_target(pose)
393     #robot_arm.set_pose_target([0.29537095654868956, 4.675568598554573e
        -05, 0.4286678926923855, 0.0017192879795506913,
        0.0014037282477544944, 0.00016120358136762693])
394     robot_plan_home = robot_arm.plan()
395 
396     robot_goal = moveit_msgs.msg.ExecuteTrajectoryGoal()
397     robot_goal.trajectory = robot_plan_home
398 
399     robot_client.send_goal(robot_goal)
400     robot_client.wait_for_result()
401     robot_arm.stop()
402 
403 def move_pose_orn(pose, arm_speed=None):
404     """Move to a given pose values, but with orientation not rpy.
405 
406     Parameters:
407     ......................
408     * pose: A Pose state object
409 
410 example of the pose state object that should be given:
411     ========================
412     position:
413         x: 0.278076372862
414         y: 0.101870353599
415         z: 0.425462888681
416     orientation:
417         x: 0.0257527874589
418         y: 0.0122083384395
419         z: 0.175399274203
420         w: 0.984084775322
421     ========================
422     """
423 
424     arm.set_pose_target(pose)
425     if arm_speed:
426         set_speed(arm_speed)
427     arm.go(wait=True)
428 
429     arm.stop()
430     arm.clear_pose_targets()
431 
432 def move_to_named_pos(position_name, arm_speed=None):
433     """Avalible names:
434     - 'resting'
435     - 'straight_forward'
436     - 'straight_up'
437     """
438     arm.set_named_target(position_name)
439     if arm_speed:
440         set_speed(arm_speed)
441     arm.go(wait=True)
```

Bibliography

[1] Robert O Ambrose et al. "Robonaut: NASA's space humanoid". In: *IEEE Intelligent Systems and Their Applications* 15.4 (2000), pp. 57–63.

[2] Tessa Brazda. *NASA Robonaut first generation*. https://www.nasa.gov/technology/r1-the-first-generation/. founded in. 2023.

[3] David Coleman et al. "Reducing the barrier to entry of complex robotic software: a moveit! case study". In: *arXiv preprint arXiv:1404.3785* (2014).

[4] Peter Corke. "Integrating ros and matlab [ros topics]". In: *IEEE Robotics & Automation Magazine* 22.2 (2015), pp. 18–20.

[5] Farbod Fahimi. *Autonomous robots*. Springer, 2009.

[6] Marc-Henri Frouin. *Niryo Company*. https://niryo.com. founded in. 2017.

[7] Marc-Henri Frouin. *Niryo Company*. https://docs.niryo.com/product/ned2/v1.0.0/en/source/introduction.html. founded in. 2022.

[8] Intel. *Types of Robots: How Robotics Technologies Are Shaping Today's World*. https://www.intel.com/content/www/us/en/robotics/types-and-applications.html. founded in. 2023.

[9] Reza N. Jazar. *Theory of Applied Robotics: Kinematics, Dynamics, and Control*. 3rd edition. Cham: Springer International Publishing, Imprint: Springer, 2022. DOI: 10.1007/978-3-030-93220-6.

[10] Charles C Kemp et al. "Humanoids". In: *experimental psychology* 56 (2009), pp. 1–3.

[11] Kevin M Lynch and Frank C Park. *Modern robotics*. Cambridge University Press, 2017. URL: www.cambridge.org/9781107156302.

[12] Alexei Makarenko, Alex Brooks, and Tobias Kaupp. "On the benefits of making robotic software frameworks thin". In: *International Conference on Intelligent Robots and Systems-Workshop for Measures and Procedures for the Evaluation of Robot Architectures and Middleware at IROS'07*. Vol. 2. 2007.

[13] Jia Pan, Sachin Chitta, and Dinesh Manocha. "FCL: A general purpose library for collision and proximity queries". In: *2012 IEEE International Conference on Robotics and Automation*. IEEE. 2012, pp. 3859–3866.

[14] Michael Peshkin and J Edward Colgate. "Cobots". In: *Industrial Robot: An International Journal* 26.5 (1999), pp. 335–341.

[15] Morgan Quigley, Brian Gerkey, and William D Smart. *Programming Robots with ROS: a practical introduction to the Robot Operating System*. " O'Reilly Media, Inc.", 2015.

[16] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. Kobe, Japan. 2009, p. 5.

[17] Open Robotics. *ROS*. https://www.ros.org/. founded in. 2015.

[18] Open Robotics. *roscore*. https://wiki.ros.org/rqt_graph. founded in. 2015.

[19] Open Robotics. *rqt_graph*. https://wiki.ros.org/rqt_graph. founded in. 2015.

[20] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[21] Ruben Smits, H Bruyninckx, and E Aertbeliën. *KDL: Kinematics and dynamics library*. https://www.orocos.org/kdl. founded in. 2011.

[22] Andrew Spielberg et al. "Functional co-optimization of articulated robots". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 5035–5042.

[23] Adam A. Stokes et al. "A Hybrid Combining Hard and Soft Robots". In: *Soft Robotics* 1.1 (2014), pp. 70–74. DOI: 10.1089/soro.2013.0002.

[24] Ioan A. Sucan and Sachin Chitta. *MoveIt*. https://moveit.ros.org/. founded in. 2013.

[25] Inc. Teradyne. *Universal Robots*. https://www.universal-robots.com/. founded in. 2005.

[26] Günter Ullrich et al. "Automated guided vehicle systems". In: *Springer-Verlag Berlin Heidelberg. doi* 10 (2015), pp. 978–3.

[27] Keenan A Wyrobek et al. "Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot". In: *2008 IEEE International Conference on Robotics and Automation*. IEEE. 2008, pp. 2165–2170.